一階述語インタプリタのアルゴリズム(数式処理と数学研究への応用)

AUTHOR(S):

元吉, 文男; 佐藤, 泰介

CITATION:

元吉, 文男 ...[et al]. 一階述語インタプリタのアルゴリズム(数式処理と数学研究への応用). 数理解析研究所講究録 1988, 646: 109-115

ISSUE DATE:

1988-02

URL:

http://hdl.handle.net/2433/100260

数理解析研究所講究録
646巻 1988年 109-115

一階述語インタプリタのアルゴリズム

電子技術総合研究所　元吉文男・佐藤泰介
(Fumio Motoyoshi and Taisuke Sato)

1．はじめに

一般に問題を記述する場合に、手続き的に記述するよりも宣言的に記述する方が自然であるし、分かりやすい。そこで数学において古くから使用されている一階述語論理で記述することが考えられるが、これはそのままプログラムにはならない。これを限定した形の式だけを許すようにしてプログラミング言語として用いたのがPrologである。

しかしPrologで許される式はホーン節と呼ばれるものであり、記述能力が弱い。ここで発表するのはホーン節よりも記述能力が高く、しかもプログラミング言語としてみた場合にも効率を落とさない言語のインタプリタである。これを以下では一階述語インタプリタと呼ぶことにする。この一階述語インタプリタで扱うことができる式は、通常のPrologのゴール以外に次の形の式を含んでいるものである：

$$\forall x : (u(x, y) \rightarrow v(x, z))$$

ただしuは、yとして変数を含まない式を与えてu(x, y)を「実行」したときにxに変数を含まない式が返ってくるように定義されているものとする。

上の式では∀の付いている変数は1個であるが、もちろん複数でも構わない。また0個の場合も許され、その場合は単なる含意となる。さらに特別な場合として後件が空である場合にはこの式はu(x, y)の否定となり、これも実行することができる。Prologにおいては否定の実行ではその中の変数が束縛されることがなく、期待した結果が得られなかったが一階述語インタプリタ

では否定されている部分の変数にも値を返すことができるようになる。

具体的には、

```
even(x) ⇔ x==0 ∨ ∃y:(x==s(y) ∧ ¬even(y)).
```

という式が定義されているときに

```
find z:even(z).
```

とすると、Prologにおけるのと同様に

```
z==0;
z==s(s(0));
z==s(s(s(s(0))));
.....
```

というように答えが得られるものである。

また、上で述べたようなゴールを使用することによって、自然な形での宣言的な形でプログラミングが行なえるようになる。たとえば、素数を求めようとしてその定義を考えてみると、それは1と自分以外に約数を持たない2以上の整数のことであるが、これを述語で書くと次のようになる。

```
prime(x) ⇔ x≧2 ∧ ∀d:( d|x → ( d=1 ∨ d=x ) )
```

これは一階述語インタプリタで直接実行できる形をしているが、現在のところは基本述語として、構文的な同等を表す「==」だけを許しているので、そのための書き換えを行なうと次のようになる：

```
prime(x) ⇔
  lessp(s(s(0)), x) ∧
  ∀d:(divide(d, x) → ( d==s(0) ∨ d==x ) ),
lessp(x, y)    ⇔ ∃z:add(x, s(z), y),
divide(x, y) ⇔
  lessp(0, x) ∧ (y==0 ∨ ∃z:(add(x, z, y) ∧ divide(x, z))),
add(x, y, z) ⇔
```

x==0 ∧ y==z ∨ ∃ p, q: (x==s(p) ∧ z==s(q) ∧ add(p, y, q)).

ここで

find x:prime(x).

とすると

x==s(s(0));

x==s(s(s(0)));

x==s(s(s(s(0))));

.....

が得られる。

2. 準備

一階述語インタプリタのアルゴリズムの説明を行なう前に、そこで使用する記号の説明、および必要な準備を述べる。

以下の説明では使用する記号によってそれが表すものを区別する。すなわち各記号はそれぞれ次の意味を持つものとする:

x, y, z	変数、
f, g	関数名、
ai, bi, e	式(iやjは添字)、
p, q, r	論理式、
==	構文的に等しいことを示す述語.

式中のある変数に代入を行なうことがあるが、これを

p{e/x}

と書くことにして、式p中の変数xに式eを代入した結果を表す。なおこのときに、e中の自由変数がp中で束縛されないことが条件であるが、インタプリタにおいて変数表の管理をうまく行なうことで実際の書き換えを避けている。

Prologでは各述語の定義はHorn節の集まりとして記述しているが、実際には定義されていないことは否定されたものと考えている場合が多く（閉世界仮説）、またここでは否定を含んだ式も正確に実行するために、ユーザにそのことを明示的に書かせることにする。これに従うとPrologのプログラム

```
add(0, Z, Z).
add(s(P), Y, s(Q)) :- add(P, Y, Q).
```

は

$$\mathrm{add}(x, y, z) \Leftrightarrow (x{=}{=}0 \wedge y{=}{=}z) \vee \exists p, q : (x{=}{=}s(p) \wedge x{=}{=}s(q) \wedge \mathrm{add}(p, y, q)).$$

と書くことになる。以後は述語はこのように定義されているものとする。

3．インタプリタのアルゴリズム

一階述語インタプリタは以下に示す変形規則に基づいて与えられた式を変形しながらプログラムの実行を行なう。

$p \Rightarrow \mathrm{unfold}(p)$ ， pがユーザ定義述語のとき ..... (R1)

$\mathrm{false} \rightarrow p \Rightarrow \mathrm{true}$ ..... (R2)

$\mathrm{true} \rightarrow p \Rightarrow p$ ..... (R3)

$(\exists x : p) \rightarrow q \Rightarrow \forall x' : (p' \rightarrow q)$ ..... (R4)

$(\forall x : p) \rightarrow q \Rightarrow \exists x' : (p' \rightarrow q)$ ..... (R5)

$(p \wedge q) \rightarrow r \Rightarrow p \rightarrow (q \rightarrow r)$ ..... (R6)

$(p \vee q) \rightarrow r \Rightarrow (p \rightarrow r) \wedge (q \rightarrow r)$ ..... (R7)

$(p \rightarrow q) \rightarrow r \Rightarrow p \wedge (q \rightarrow r) \vee r$ ..... (R8)

$x{=}{=}x \rightarrow q \Rightarrow q$ ..... (R9)

$\forall x : (x{=}{=}e \rightarrow q) \Rightarrow \mathrm{true}$ ， xがe中にあるとき

=> q{e/x} , それ以外 ..... (R10)

∀y1:...:∀yn:(x==e →q)

=> (∀y1:...:∀yn:¬x==e) ∨ (∃y1:...:∃yn:(x==e∧q)),

e中の全称限量された全ての変数をy1, y2,... とする..... (R11)

x==x => true ..... (R12)

x==e => false, xがe中にあるとき. ... (R13)

∃y:(y==e ∧ q) => q{e/y} ,

yがe中に自由変数として現われないとき ..... (R14)

f(a1,...,am)==f(b1,...,bm) => a1==b1 ∧ ... ∧ am==bm .... (R15)

f(a1,...,am)==g(b1,...,bn) => false ,

fとgが異なり、mとnが異なるとき ..... (R16)

∃x:(p ∨ q) => (∃x:p) ∨ (∃x:q) ..... (R17)

∀x:(p ∧ q) => (∀x:p) ∧ (∀x:q) ..... (R18)

¬p => p→false , if p is not a form 'x==e' ..... (R19)

false∧p => false ..... (R20)

false∨p => p ..... (R21)

true∧p => p ..... (R22)

true∨p => true ..... (R23)

以上の規則はR10・R11を除いては容易に導けるものである。R10・R11はさらに一般的な次の規則の特別な例であり、この規則が一階述語インタプリタの要でもある：

∀x:(u(x, y)→v(x, z))

⇔ ∀x:¬u(x, y) ∨ ∃x:(u(x, y)∧v(x, z)).

ここで、任意のyについてu(x, y)を満たすxは高々1個しかないものとする。また、この式におけるxは複数でも差しつかえない。

4．例

以上で説明した一階述語インタプリタをLisp上に作成してみたが、以下にその例を示す。

最初は否定を含んだゴールに、未束縛の変数があっても正しく実行できる例である。

```
=> (define (even x)
     (or (== x (o))
         (exist (y)
           (and (== x (s y)) (not (even y))))))
(EVEN X)
=> (find (a) (even a))
A = (O);
A = (S (S (O)));
A = (S (S (S (S (O)))))
```

次の例は、素数の（宣言的な）定義を与えて素数を求めるプログラムである。

```
=> (define (prime x)
     (and (exist (y) (add y (s (s (o))) x))
          (any (d) (imply (div d x) (or (== d (s (o))) (== d x))))))
(PRIME X)
=> (define (div x y)
     (exist (z)
       (and (== x (s z))
            (or (== y (o))
                (exist (p) (and (add x p y) (div x p)))))))
```

```
(DIV X Y)
=> (define (add x y z)
     (or (and (== x (o)) (== y z))
         (exist (p q)
           (and (== x (s p)) (and (== z (s q)) (add p y q))))))
(ADD X Y Z)
=> (find (x) (prime x))
X = (S (S (O)));
X = (S (S (S (O))));
X = (S (S (S (S (S (O))))));
X = (S (S (S (S (S (S (S (O))))))));
X = (S (S (S (S (S (S (S (S (S (S (O))))))))))
```

5. おわりに

一階述語インタプリタを変形規則を用いて実現する方法を述べたがこの変形を実際に行なうのでなく、環境表を利用してポインタの付け替えで行なうことによって、必要な操作の数はoccur checkを行なう純Prologと同程度で済ますことができる。

現在のところでは、基本的な述語は構文的同値を表す「==」だけであり、実用的なプログラミング言語としてはまだ使用できないが、他の述語も含んだ形で拡張を行なうことにより、一階述語で問題を記述するのに適した分野（たとえば数式処理など）で使用することが可能になると思われる。